PAPER *Special Section on Data Engineering and Information Management*

Satisfiability of Simple XPath Fragments under Duplicate-Free DTDs*

Nobutaka SUZUKI[†a)], ***Member***, **Yuji FUKUSHIMA**[††], ***and*** **Kosetsu IKEDA**[†††], ***Nonmembers***

SUMMARY In this paper, we consider the XPath satisfiability problem under restricted DTDs called "duplicate free". For an XPath expression q and a DTD D, q is *satisfiable* under D if there exists an XML document t such that t is valid against D and that the answer of q on t is nonempty. Evaluating an unsatisfiable XPath expression is meaningless, since such an expression can always be replaced by an empty set without evaluating it. However, it is shown that the XPath satisfiability problem is intractable for a large number of XPath fragments. In this paper, we consider simple XPath fragments under two restrictions: (i) only a label can be specified as a node test and (ii) operators such as qualifier ([ ]) and path union (∪) are not allowed. We first show that, for some small XPath fragments under the above restrictions, the satisfiability problem is NP-complete under DTDs without any restriction. Then we show that there exist XPath fragments, containing the above small fragments, for which the satisfiability problem is in PTIME under duplicate-free DTDs.

key words: *XML, XPath, satisfiability*

1. Introduction

XPath has been a common query language for XML, and several query/transformation languages such as XSLT and XQuery are also based on XPath. For an XPath expression q and a DTD D, q is *satisfiable* under D if there exists an XML document t such that t is valid against D and that the answer of q on t is nonempty. Evaluating an unsatisfiable XPath expression is meaningless, since such an expression can always be replaced by an empty set without evaluating it. However, it is shown that the satisfiability problem is intractable for a large number of XPath fragments [1], [2]. Therefore, it is important to find XPath fragments for which the satisfiability problem can be solved efficiently.

Let us show a simple example of an unsatisfiable XPath expression. Consider the following DTD.

```
<!ELEMENT students (undergraduate|graduate)+>
<!ELEMENT undergraduate (name,email)>
<!ELEMENT graduate (name,email,supervisor?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT supervisor (#PCDATA)>
```

Let q = `//supervisor/parent` :: `undergraduate/name` be an XPath query. Then q would return the names of undergraduate students that have a supervisor. However, it is easy to see that q is unsatisfiable since an `undergraduate` element cannot have any `supervisor` element as a child. Clearly, we should detect unsatisfiable XPath expressions prior to evaluating them.

Although the above example is quite simple, current XML documents and schemas are becoming very large and complex. Thus, when a user obtains an empty result of a query, it is often difficult for the user to tell whether the query is unsatisfiable or it is satisfiable but the target XML document happens to have no answer to the query. If we have an efficient algorithm for the XPath satisfiability problem, the user can easily tell whether he/she has to correct the query.

In this paper, we focus on simple XPath fragments using child (↓), descendant-or-self ($\downarrow^*$), parent (↑), following-sibling ($\rightarrow^+$), and preceding-sibling ($\leftarrow^+$) axes under two restrictions; (i) only a label can be specified as a node test and (ii) operators such as qualifier ([ ]) and path union (∪) are not allowed. We first consider two simple XPath fragments $XP^{\{\downarrow,\uparrow\}}$ and $XP^{\{\downarrow,\rightarrow^+,\leftarrow^+\}}$, where $XP^{\{\downarrow,\uparrow\}}$ stands for the XPath fragments using only child and parent axes under the above restrictions. We show that, even for these small fragments, the satisfiability problem is NP-complete under DTDs without any restriction. We show on the other hand that, under duplicate-free DTDs, satisfiability for $XP^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ can be solved in polynomial time. Here, a DTD D is *duplicate free* if no content model of D uses the same label more than once, and most real-world DTDs are duplicate free [3]. Then we consider incorporating descendant-or-self axis. We show that satisfiability for $XP^{\{\downarrow,\uparrow,\downarrow^*\}}$ is NP-complete even under duplicate-free DTDs. On the other hand, we show that satisfiability for $XP^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ can be solved in polynomial time under duplicate-free DTDs if an XPath expression contains only a constant number of descendant-or-self axes.

Beyond the earlier work [1], [2], several other eminent studies have investigated the XPath satisfiability problem. Hidders considered the XPath satisfiability problem without DTD [4]. Lakshmanan et al. considered the satisfia-

Manuscript received July 2, 2012.
Manuscript revised October 29, 2012.

[†]The author is with the Faculty of Library, Information and Media Studies, University of Tsukuba, Tsukuba-shi, 305–8550 Japan.
[††]The author is with Yahoo Japan Corporation, Minato-ku, Tokyo, 107–6211 Japan.
[†††]The author is with the Graduate School of Library, Information and Media Studies, University of Tsukuba, Tsukuba-shi, 305–8550 Japan.

[*]This work is based on an eariler work: Satisfiability of simple xpath fragments in the presence of dtds, in Proceeding of the eleventh international workshop on Web information and data management (WIDM'09) ©ACM, 2009. http://doi.acm.org/10.1145/1651587.1651594

a) E-mail: nsuzuki@slis.tsukuba.ac.jp

DOI: 10.1587/transinf.E96.D.1029

bility problem for tree pattern queries with and without DTDs [5]. Their tree patten queries and the XPath fragments in this paper are incomparable, e.g., the former supports node equalities but the latter does not, while the former does not fully capture following-sibling and preceding-sibling axes. Montazerian et al. proposed two classes of restricted DTDs, duplicate-free DTDs and covering DTDs, and they showed that satisfiability for several XPath fragments can be solved in polynomial time under these DTDs [3]. Their XPath fragments support qualifier, union, and wildcard node test, but not parent, following-sibling, or preceding-sibling axis. Figueira investigated satisfiability for $XP^{\{\downarrow,\downarrow^*,=\}}$ without DTD and showed that the problem is EXPTIME-complete [6]. Ishihara et al. proposed subclasses of covering DTDs and investigated the tractability of XPath satisfiability under the subclasses [7], [8]. Finally, this paper is a revised version of Ref. [9]. This paper provides (i) proofs of the correctness of our algorithms, (ii) a proof of the running time of the algorithm for $XP^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ and (iii) a discussion relaxing the restriction to our XPath fragment, as well as a number of revisions to the original version.

The rest of this paper is organized as follows. Section 2 gives some preliminaries. Section 3 shows the NP-completeness of satisfiability for $XP^{\{\downarrow,\uparrow\}}$ and $XP^{\{\downarrow,\rightarrow^+,\leftarrow^+\}}$. Section 4 presents a polynomial-time algorithm that solves satisfiability for $XP^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ under duplicate-free DTDs. Section 5 shows the NP-completeness of satisfiability for $XP^{\{\downarrow,\uparrow,\downarrow^*\}}$. Section 6 presents an algorithm that solves satisfiability for $XP^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ under duplicate-free DTDs. Section 7 summarizes the paper.

2. Definitions

An XML document is modeled as a node-labeled ordered tree (attributes are omitted). A text node is omitted, in other words, we assume that each leaf node has a text node implicitly. For a node n in a tree, by $l(n)$ we mean the label of n, representing the element name of n. In what follows, we use the term tree when we mean node-labeled ordered tree.

Let Σ be a set of labels. Then a *regular expression* over Σ is defined as follows.

- ϵ and a are regular expressions, where $a \in \Sigma$.
- Let $e_1, e_2, \cdots, e_n$ be regular expressions. Then $r = (e_1 e_2 \cdots e_n)$ and $r' = (e_1|e_2|\cdots|e_n)$ are regular expressions. Each e_i is a *subexpression* of r and r'.
- Let e be a regular expression. Then $r = e^*$ is a regular expression. e is a *subexpression* of r.

For sets L, L' of strings over Σ, the *concatnation* of L and L' is defined as $LL' = \{w_1 w_2 \mid w_1 \in L, w_2 \in L'\}$. Then $L^0 = \{\epsilon\}$ and $L^i = L^{i-1}L$ for $i \geq 1$. The *language* of a regular expression r, denoted $L(r)$, is defined as follows.

- $L(\epsilon) = \{\epsilon\}$ and $L(a) = \{a\}$, where $a \in \Sigma$.
- Let $e_1, e_2, \cdots, e_n$ be regular expressions. Then $L(e_1 e_2 \cdots e_n) = L(e_1)L(e_2)\cdots L(e_n)$ and $L(e_1|e_2|\cdots|e_n) = L(e_1) \cup L(e_2) \cup \cdots \cup L(e_n)$.
- Let e be a regular expression. Then $L(e^*) = \bigcup_{i\geq 0} L(e)^i$.

A *DTD* is a tuple $D = (d, s)$, where d is a mapping from Σ to the set of regular expressions over Σ and $s \in \Sigma$ is the *start label*. For a label $a \in \Sigma$, $d(a)$ is the *content model* of a. A tree t is *valid* against D if (i) the root of t is labeled by s and (ii) for each node n in t $l(n_1)\cdots l(n_m) \in L(d(l(n)))$, where $n_1 \cdots n_m$ are the children of n. Let r be a regular expression and $\Sigma(r)$ be the set of labels appearing in r. Then r is *duplicate free* if each label in $\Sigma(r)$ occurs exactly once in r. A DTD D is *duplicate free* if for each content model $d(a)$ of D, $d(a)$ is duplicate free. For example, let $D = (d, s)$, where $d(s) = (a^*b)|a?$ and $d(a) = d(b) = \epsilon$. Then D is not duplicate free due to $d(s)$.

A *location step* is of the form $axis :: l$, where (i) $axis$ is either $\downarrow$ (the child axis), $\downarrow^*$ (the descendant-or-self axis), $\uparrow$ (the parent axis), $\rightarrow^+$ (the following-sibling axis), or $\leftarrow^+$ (the preceding-sibling axis), and (ii) l is a label. An *XPath query* (*query* for short) is $/ls_1/ls_2\cdots/ls_n$, where ls_i is a location step. Let XP be the set of XPath queries. We denote a fragment of XP by listing the axes supported by the fragment. For example, $XP^{\{\downarrow,\downarrow^*\}}$ denotes the set of queries using only child and descendant-or-self axes.

Let t be a tree and q be a query. We say that t *satisfies* q, denoted $t \vDash q$, if the answer of q on t is nonempty. If there is a tree t such that t is valid against a DTD D and that $t \vDash q$, then q is *satisfiable* under D. For an XPath fragment XP^S, the *XPath satisfiability problem* for XP^S, denoted $SAT(XP^S)$, is to decide, for a DTD D and a query $q \in XP^S$, if q is satisfiable under D.

3. Simple XPath Fragments for which Satisfiability is Intractable

In this section, we show that the XPath satisfiability problem is NP-complete for two simple XPath fragments under DTDs without any restrictions.

Ref. [1] shows that $SAT(XP^{\{\downarrow,\uparrow\}})$ is NP-complete if a wildcard is allowed as a node test. The following theorem shows a slightly more strong result; $SAT(XP^{\{\downarrow,\uparrow\}})$ is NP-complete even if only a label is allowed as a node test.

Theorem 1: $SAT(XP^{\{\downarrow,\uparrow\}})$ is NP-complete.

Proof: For a query q and a tree t valid against a DTD, it can be determined in polynomial time whether the answer of q on t is nonempty [10]. Thus the problem is in NP.

To show that the problem is NP-hard, we reduce 3SAT to the XPath satisfiability problem. Let

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

be an instance of 3SAT, where C_i is a clause consisting of three literals. Let $x_1, x_2, \cdots, x_m$ be the variables appearing in ϕ. From this instance we construct an instance of the XPath satisfiability problem.

Let $\Sigma = \{s\} \cup \{c_1, c_2, \cdots, c_n\}$ ($c_i \neq c_j$ whenever $i \neq j$). DTD $D = (d, s)$ is defined as follows.

$$d(s) = (T_1|F_1)(T_2|F_2)\cdots(T_m|F_m),$$

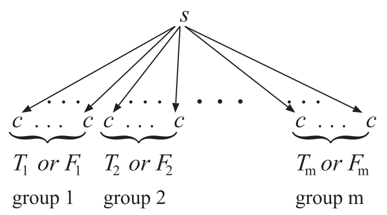


Fig. 1 A tree used in the proof of Theorem 1.

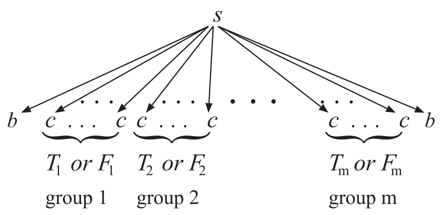


Fig. 2 A tree used in the proof of Theorem 2.

$$d(c_i) = \epsilon, \quad (1 \le i \le n)$$

where label $c_i \in \Sigma$ corresponds to clause C_i, and T_i and F_i stand for sequences of labels defined as follows.

- T_i represents the clauses in ϕ that contain positive literal x_i. That is, if $C_{i1}, \cdots, C_{ik}$ are the clauses containing positive literal x_i, then $T_i = c_{i1} \cdots c_{ik}$. In other words, T_i consists of the clauses that become true by setting $x_i = true$.
- F_i represents the clauses in ϕ that contain negative literal $\neg x_i$. That is, if $C_{i1}, \cdots, C_{ik}$ are the clauses containing negative literal $\neg x_i$, then $F_i = c_{i1} \cdots c_{ik}$.

Figure 1 presents a tree valid against D. Then query q is defined as follows.

$$\begin{aligned} q = {} & /{\downarrow}{::}\, s/{\downarrow}{::}\, c_1 \\ & /{\uparrow}{::}\, s/{\downarrow}{::}\, c_2/{\uparrow}{::}\, s/{\downarrow}{::}\, c_3/\cdots/{\uparrow}{::}\, s/{\downarrow}{::}\, c_n \end{aligned} \tag{1}$$

Thus, $t \vDash q$ if the root of t is labeled by s and has a child labeled by c_i for every $1 \le i \le n$. In the following, we show that ϕ is satisfiable *iff* q is satisfiable under D.

Only if part: Assume that ϕ is satisfiable. Then there is a truth assignment α for $x_1, \cdots, x_m$ satisfying ϕ. By using α we construct a tree t valid against D as follows.

- For every $1 \le i \le m$, if $\alpha(x_i) = true$, then the ith group in t is set to T_i (Fig. 1), otherwise the ith group is set to F_i.

Since all clauses $C_1, \cdots, C_n$ become true under α, the root s has at least one child labeled by c_i for every $1 \le i \le n$. Hence $t \vDash p$.

If part: Assume that q is satisfiable under D. Then there is a tree t valid against D such that the root of t is labeled by s and has a child labeled by c_i for every $1 \le i \le n$. Let α be a truth assignment defined as follows ($1 \le i \le m$).

$$\alpha(x_i) = \begin{cases} true & \text{if the } i\text{th group matches } T_i, \\ false & \text{if the } i\text{th group matches } F_i. \end{cases}$$

Since the root s has a child labeled by c_i for every $1 \le i \le n$, clauses $C_1, \cdots, C_n$ become true under α. □

Without upward axis, the satisfiability problem is NP-complete if both $\rightarrow^+$ and $\leftarrow^+$ are allowed.

Theorem 2: SAT(XP$^{\{\downarrow,\rightarrow^+,\leftarrow^+\}}$) is NP-complete.

Proof: We can show that the problem is in NP similarly to Theorem 1. We show that this problem is NP-hard by a reduction from 3SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ be an instance of 3SAT, and let $x_1, x_2, \cdots, x_m$ be the variables occurring in ϕ. Without loss of generality, we assume that n is an even number. From this instance, we construct an instance of the XPath satisfiability problem.

DTD $D = (d, s)$ is defined as follows.

$$\begin{aligned} d(s) &= b(T_1|F_1)(T_2|F_2)\cdots(T_m|F_m)b, \\ d(b) &= \epsilon, \\ d(c_i) &= \epsilon, \quad (1 \le i \le n) \end{aligned}$$

where T_i and F_i are defined similarly to Theorem 1. Figure 2 shows a tree t valid against D, where the leftmost and rightmost nodes labeled by b are "boundary" nodes. Query q is defined as follows.

$$q = /\ {\downarrow}{::}\ s/\ {\downarrow}{::}\ b \tag{2}$$
$$/ \rightarrow^+{::}\ c_1/ \rightarrow^+{::}\ b \tag{3}$$
$$/ \leftarrow^+{::}\ c_2 :: / \leftarrow^+{::}\ b \tag{4}$$
$$\vdots$$
$$/ \rightarrow^+{::}\ c_{n-1}/ \rightarrow^+{::}\ b$$
$$/ \leftarrow^+{::}\ c_n$$

For a tree t valid against D, q checks if the root of t has a child labeled by c_i for every $1 \le i \le n$, as follows.

1. By line (2), goes down to the left boundary node.
2. By line (3), moves to right and find a node labeled by c_1, then goes to the right boundary node.
3. By line (4), moves to left and find a node labeled by c_2, goes to the left boundary node, and so on.

Now we can show that ϕ is satisfiable *iff* p is satisfiable under D, similarly to Theorem 1. □

Thus, under DTDs without any restrictions the XPath satisfiability problem is unlikely to be solved efficiently even for the above simple XPath fragments. In the next section, we show an XPath fragment, containing the above fragments, for which satisfiability can be solved in PTIME under duplicate-free DTDs.

4. Algorithm for XPath Fragment without Descendant-or-Self Axis under Duplicate-Free DTDs

In this section, we present a polynomial-time algorithm for solving SAT(XP$^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$) under duplicate-free DTDs.

Before presenting the algorithm formally, we give a preliminary definition. Let q be a query. A *traverse tree* of q is a tree representing the "walking path" of q such that each node n has the set $I(n)$ of indexes of the location steps accessing n. For example, Fig. 3 presents a traverse tree of $q = /a/b/c/ \uparrow\!:: b/d/e/ \rightarrow^+\!:: f$, where $I(n_1) = \{1\}$, $I(n_2) = \{2,4\}$, $I(n_3) = \{5\}$, $I(n_4) = \{6\}$, $I(n_5) = \{7\}$, $I(n_6) = \{3\}$. Formally, a *traverse tree* of $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ is defined as follows.

- The case where $|q| = 1$: Let $q = /ax :: lb$. If $ax = \downarrow$, then an edge $n' \rightarrow n$ such that $l(n') = root$, $l(n) = lb$, $I(n') = \emptyset$, and that $I(n) = \{1\}$ is the traverse tree of q.
- The case where $|q| > 1$: Let $q = q'/ax :: lb$, where q' is a query with $|q'| = |q| - 1$. Let t be a traverse tree of q' and n be the "context node" in t, i.e., $|q'| \in I(n)$. We have four cases according to ax.

  1. The case where $ax = \downarrow$:

     a. If n has a child n' labeled by lb, then the tree obtained from t by adding $|q|$ to $I(n')$ is a traverse tree of q, or

     b. A tree obtained from t by adding a new node n' with $I(n') = \{|q|\}$ as a child of n is a traverse tree of q.

  2. The case where $ax = \uparrow$:

     a. If the parent n' of n is labeled by lb, then the tree obtained from t by adding $|q|$ to $I(n')$ is a traverse tree of q.

  3. The case where $ax = \rightarrow^+$:

     a. If n has a right sibling n' labeled by lb, then the tree obtained from t by adding $|q|$ to $I(n')$ is a traverse tree of q, or

     b. The tree obtained from t by adding a new node n' with $I(n') = \{|q|\}$ and $l(n') = lb$ as a right sibling of n is a traverse tree of q.

  4. The case of $ax = \leftarrow^+$ is defined similarly to Case (3).

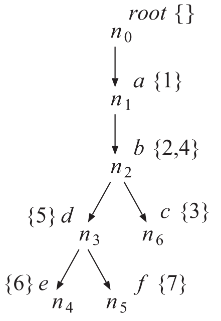


Fig. 3 An example of traverse tree.

Let t be a traverse tree. A *supertree* of t is recursively defined as follows.

- Let t' be the tree obtained by dropping $I(n)$ for each node n in t. Then t' is a supertree of t.
- Let t' be a supertree of t. Then a tree obtained by adding a leaf node n with $l(n) \in \Sigma$ to t is a supertree of t.

We say that a traverse tree t of q is *valid* against D if there is a supertree t' of t such that t' is valid against D and that t and t' share the same root and its child.

In short, for a query q and a duplicate-free DTD D, the algorithm constructs, along the location steps of q, a traverse tree of q valid against D. The algorithm returns "unsatisfiable" if D is violated during constructing a traverse tree. If no location step of q violates D, the algorithm returns "satisfiable". Let us give an example (Fig. 4). Let

$$q = / \downarrow\!:: a/ \downarrow\!:: b/ \uparrow\!:: a/ \downarrow\!:: c/ \rightarrow^+\!:: d/ \rightarrow^+\!:: e$$

be a query and $D = (d, a)$ be a DTD, where $d(a) = bc(d|e)$ and $d(b) = d(c) = d(d) = d(e) = \epsilon$. The algorithm first creates a single node labeled by *root* (Fig. 4 (a)) and then modifies it step-by-step. Each tree in Fig. 4 represents the traverse tree constructed by the location steps encountered so far. For example, in Fig. 4(a,b), according to location step "$\downarrow\!:: a$" a node n_1 labeled by a is inserted as the child of n_0. Each tree has one marked node that represents the "context node" in the tree. In Fig. 4, each marked node is circled. When the algorithm encounters a location step violating D, then the algorithm returns "unsatisfiable". For example, in Fig. 4 (f) according to location step "$\rightarrow^+\!:: e$" the algorithm tries to insert a node labeled by e as the right sibling of n_4, but it is impossible to insert such a node due to the definition of $d(a)$. Hence "unsatisfiable" is returned.

Now we present the "main" algorithm. Each subroutine in lines 5 to 11 (shown later) modifies input tree t according to location step axis[i] :: label[i] and returns the modified tree. If the location step violates D, then *nil* is returned.

Input: A query q=/axis[1] :: label[1]/ $\cdots$ /axis[m] :: label[m]

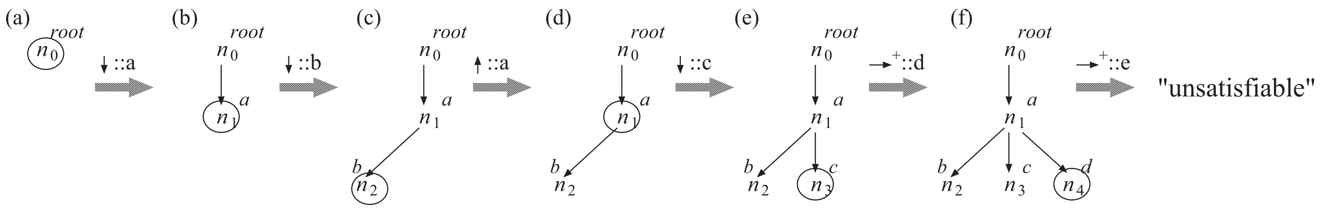


Fig. 4 Trees created by the algorithm in Sect. 4.

```
         and a duplicate-free DTD $D = (d, s)$.
Output: “satisfiable” or “unsatisfiable”.
  begin
 1.  Create a node $n$ labeled by “root”.
     Let $t$ be the tree consisting only of $n$.
     Assume that $d(\mathrm{root}) = s$.
 2.  Mark $n$.
 3.  for $i = 1$ to $m$ do
 4.     if axis[$i$] = ‘$\downarrow$’ then
 5.        $t \leftarrow$ do_child($t$, label[$i$]);
 6.     else if axis[$i$] = ‘$\uparrow$’ then
 7.        $t \leftarrow$ do_parent($t$, label[$i$]);
 8.     else if axis[$i$] = ‘$\rightarrow^+$’ then
 9.        $t \leftarrow$ do_following-sibling($t$, label[$i$]);
10.     else if axis[$i$] = ‘$\leftarrow^+$’ then
11.        $t \leftarrow$ do_preceding-sibling($t$, label[$i$]);
12.     end
13.     if $t = nil$ then
14.        return “unsatisfiable”;
15.     end
16.  end
17.  return “satisfiable”;
  end
```

In the following, we present the subroutines. First, to present do_child, we need a definition. Let n be a node in t with children $n_1, \cdots, n_k$ and l be a label. We say that a node labeled by l is *insertable* as a child of n if there are words w_1, w_2 such that $w_1 l w_2 \in L(d(l(n)))$ and that $w_1 w_2$ is a supersequence of $l(n_1) \cdots l(n_k)$. In line 4 below, “mark n'” means that the mark on n is moved to n', since a tree has exactly one marked node at all times.

do_child(t, l)

```
  begin
 1.  Let $n$ be the marked node in $t$.
 2.  if a node labeled by $l$ is insertable
     as a child of $n$ then
 3.     Add a new node $n'$ labeled by $l$ as a child of $n$.
 4.     Mark $n'$.
 5.     return $t$;
 6.  else if $n$ has a child $n'$ labeled by $l$ then
 7.     Mark $n'$.
 8.     return $t$;
 9.  else
10.     return $nil$;
11.  end
  end
```

The order of the **if** statements on lines 2 and 6 is significant, since if the order of the two **if** statements are exchanged, then the correctness of the algorithm cannot be guaranteed. Actually, the proof of the correctness of the algorithm heavily depends on the fact that the algorithm creates a new node whenever possible, as shown in the proof of Lemma 2 given later.

Let us present how to check if a node labeled by l is insertable as a child of n. Let n be a node, $n_1, \cdots, n_k$ be the children of n, and l be a label. Then a node labeled by l is insertable as a child of n *iff* $k = 0$ and l appears in $d(l(n))$, or, $k \geq 1$ and one of the following two conditions holds for every $1 \leq i \leq k$ (the proof of the correctness of this condition is omitted).

1. $d(l(n))$ contains a subexpression $(e_1 e_2 \cdots e_h)$ such that for some $f, g \in \{1, 2, \cdots, h\}$ with $f \neq g$ e_f contains $l(n_i)$ and e_g contains l.
2. $d(l(n))$ contains a subexpression e^* such that e contains l and $l(n_i)$.

Intuitively, the above two condition mean that l and $l(n_i)$ can “coexist” for $1 \leq i \leq k$, i.e., there is a string in $L(d(l(n))$ containing l as well as $l(n_1), l(n_2), \cdots, l(n_k)$.

Here, let us briefly show the reason why we cannot determine efficiently whether a node labeled by l is insertable as a child of n under non-duplicate-free DTDs. Since an XML tree is an ordered tree, we have to determine the order of the children of n. If $d(l(n))$ is duplicate free, then it is easy to determine such an order since for each child n_i, $d(l(n))$ contains at most one label that coincides with $l(n_i)$ and thus the position of n_i can be determined easily. On the other hand, if $d(l(n))$ is not duplicate free, then we have to find an appropriate order of the permutations of $\{l(n_1), l(n_2), \cdots, l(n_k), l\}$, which cannot be solved efficiently. For example, consider the tree in Fig. 1. The order of the leaf nodes visited by q defined in (1) is a permutation of $\{c_1, c_2, \cdots, c_n\}$, which cannot be found efficiently by Theorem 1.

Second, do_parent can be defined easily, as follows.

do_parent(t, l)

```
  begin
 1.  Let $n$ be the marked node in $t$.
 2.  if $n$ has no parent or
     the parent of $n$ is not labeled by $l$ then
 3.     return $nil$;
 4.  else
 5.     Mark the parent of $n$.
 6.     return $t$;
 7.  end
  end
```

Finally, let us present do_following-sibling (do_preceding-sibling is defined similarly). Let n be a node with children $n_1, \cdots, n_k$ and l be a label. We say that a node labeled by l is *insertable* as a right sibling of n_i if there are words w_1, w_2 such that $w_1 l w_2 \in L(d(l(n)))$ and that for some $j \geq i$ w_1 is a supersequence of $l(n_1) \cdots l(n_j)$ and w_2 is a supersequence of $l(n_{j+1}) \cdots l(n_k)$.

do_following-sibling(t, l)

```
  begin
 1.  Let $n$ be the marked node in $t$.
 2.  if a node labeled by $l$ is insertable
     as a right sibling of $n$ then
 3.     Add a new node $n'$ labeled by $l$ as a right sibling
        of $n$.
 4.     Mark $n'$.
 5.     return $t$;
 6.  else if $n$ has a right sibling $n'$ labeled by $l$ then
 7.     Mark $n'$;
 8.     return $t$;
 9.  else
10.     return $nil$;
11.  end
  end
```

Similarly to do_child, the order of the **if** statements on liens 2 and 6 cannot be exchanged. In line 2, whether a node labeled by l is insertable as a right sibling of n can be checked in a similar manner to check if a node labeled by l is insertable as a child of n, used in do_child (since D is duplicate free, whether label l can be a right sibling of l' can be checked easily).

In the following, we show the correctness and the time complexity of the algorithm. First, we show the correctness of the algorithm, as follows.

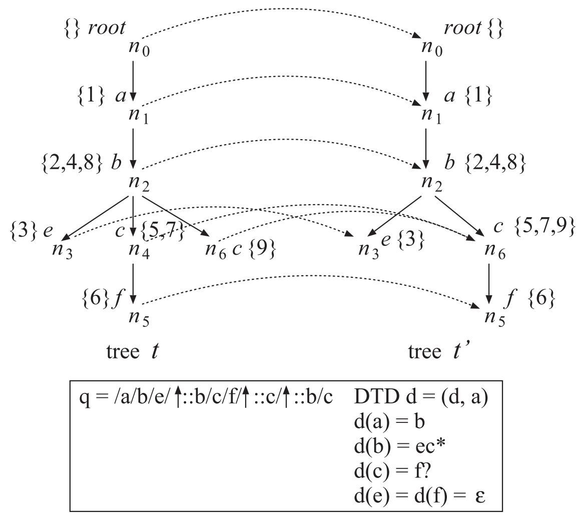
1. We show that, for a query $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ and a duplicate-free DTD D, there is a traverse tree of q valid against D *iff* q is satisfiable under D (Lemma 1).
2. We show that any traverse tree t of q valid against D is obtained by "folding" the traverse tree of q constructed by the algorithm (Lemma 2).
3. By using the two lemmas, we show that the algorithm constructs a traverse tree of q valid against D (i.e., the algorithm returns "satisfiable") *iff* q is satisfiable under D (Theorem 3).

Lemma 1: Let $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ be a query and D be a duplicate-free DTD. Then there is a traverse tree of q valid against D *iff* q is satisfiable under D.

Proof (sketch): The only if part holds trivially. Suppose that q is satisfiable under D. Then there is a tree t valid against D such that the answer of q is nonempty. Thus there must be a "walking path" of q on t, which represents a traverse tree of q valid against D. □

Lemma 2: Let D be a duplicate-free DTD, $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ be a query, and t be the value of "variable t" in line 13 of the "main" algorithm for (q, D). Then we have

1. t is a traverse tree of q valid against D whenever $t \neq nil$, and
2. if there is a traverse tree t' of q valid against D, then $t \neq nil$ and there is a total and surjective function $h : N \rightarrow N'$ satisfying the following condition, where N and N' are the sets of nodes of t and t', respectively (Fig. 5 presents an example of function h by dashed arcs between t and t').
   - For every node n' in t', $I(n') = I(n_i) \cup \cdots \cup I(n_j)$, where $n_i, \cdots, n_j$ is the nodes in t such that $h(n_i) = \cdots = h(n_j) = n'$.



q = /a/b/e/↑::b/c/f/↑::c/↑::b/c

DTD d = (d, a)
d(a) = b
d(b) = ec*
d(c) = f?
d(e) = d(f) = ε

Fig. 5 An example of function h.

Proof (sketch): Condition (1) follows from the construction of the algorithm. Condition (2) follows from the following observation.

- do_child, do_following-sibling, and do_preceding-sibling creates a new node whenever possible, but
- during constructing a traverse tree, a new node may not be created even if it is possible, that is, Cases (1-a) and (3-a) of the definition may be selected instead of Cases (1-b) and (3-b), when $\downarrow$ and $\rightarrow^+$ axes are encountered, respectively.

□

We now have the following theorem.

Theorem 3: Let $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$ be a query and D be a duplicate-free DTD. Then the algorithm returns "satisfiable" *iff* q is satisfiable under D.

Proof: If the algorithm returns "satisfiable", then by the construction of the algorithm we can show that the tree created by the algorithm for (q, D) is a traverse tree valid against D.

In the following, we show that if the algorithm returns "unsatisfiable", then q is unsatisfiable under D. Suppose that the algorithm returns "unsatisfiable". This implies that there is no traverse tree of q valid against D. Suppose contrarily that the algorithm returns "unsatisfiable" but that there is a traverse tree t' of q valid against D. Then the algorithm constructs a tree $t \neq nil$ such that Condition (2) of Lemma 2 holds for t and t'. But this is a contradiction since the algorithm returns "unsatisfiable", i.e., the value of t in line 13 of the "main" algorithm must be nil. Hence q is unsatisfiable under D by Lemma 1. □

Then we show the complexity of the algorithm.

Theorem 4: The algorithm runs in $O(|q|(|q| + |D|))$ time, where $|q|$ denotes the number of location steps in q and $|D|$ is the description length of D.

Proof (sketch): Let t be the tree created by the algorithm. The size of t is in $O(|q|)$. For each node n in t, the running time of do_{child, parent, following-sibling, preceding-sibling} is in $O(|q| + |D|)$. □

5. XPath Fragment with Descendant-or-Self Axis

In this and the next sections, we consider XPath fragments with descendant-or-self axis. In this section, we show that $\mathrm{SAT}(\mathrm{XP}^{\{\downarrow,\uparrow,\downarrow^*\}})$ is NP-complete even under duplicate-free DTDs.

Theorem 5: $\mathrm{SAT}(\mathrm{XP}^{\{\downarrow,\uparrow,\downarrow^*\}})$ is NP-complete under duplicate-free DTDs.

Proof: We can show that the problem is in NP similarly to Theorem 1. We show the NP-hardness of the problem by reducing 3SAT to this problem. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ be an instance of 3SAT, and let $x_1, x_2, \cdots, x_m$ be the variables occurring in ϕ. Without loss of generality, we assume that for any $1 \leq i \leq n$ and any $1 \leq j \leq m$, C_i does not contain both positive literal x_j and negative literal $\neg x_j$ at the same time.

From ϕ we construct an instance of the XPath satisfiability problem. First, DTD $D = (d, s)$ is constructed as follows.

$$\begin{aligned}
d(s) &= c \\
d(c) &= (c|e)x \\
d(e) &= \epsilon \\
d(x) &= y(x|b) \\
d(y) &= (y|b')(c_1|\cdots|c_{n+1}) \\
d(b) &= \epsilon \\
d(b') &= \epsilon \\
d(c_i) &= \epsilon \quad (1 \leq i \leq n+1)
\end{aligned}$$

It is clear that D is duplicate free.

Query q is a sequence of the following $2 + 2m + n$ subqueries.

$$q = q_1\, q_{T_1} q_{F_1} \cdots q_{T_m} q_{F_m}\, q_2\, q_{check_1} \cdots q_{check_n}.$$

In brief, q checks the satisfiability of ϕ as follows. First, $q_1\, q_{T_1} q_{F_1} \cdots q_{T_m}$ "constructs" a tree t presented in Fig. 6. Then $q_2\, q_{check_1} \cdots q_{check_n}$ checks the satisfiability of ϕ over t.

Let us give the subqueries of q. First, q_1 is defined as follows. q_1 traces subtree (A) of Fig. 6.

$$q_1 = /s/c/c/e/\uparrow\!::c/\uparrow\!::c/\uparrow\!::s.$$

Then we define q_{T_i} and q_{F_i} $(1 \leq i \leq m)$. q_{T_i} is defined as follows.

$$q_{T_i} = /\downarrow^*\!::c \tag{5}$$

$$\underbrace{/x/ \;\cdots\; /x}_{i\,\text{location steps}} \tag{6}$$

$$/y/c_{i_1}/\uparrow\!::y/y/c_{i_2}/\uparrow\!::y/\cdots/y/c_{i_n}/\uparrow\!::y/b' \tag{7}$$

$$\underbrace{/\uparrow\!::y/\cdots/\uparrow\!::y}_{n\text{ location steps}} \tag{8}$$

$$\underbrace{/\uparrow\!::x/\cdots/\uparrow\!::x}_{i\text{ location steps}} \tag{9}$$

$$/\uparrow\!::c/\downarrow^*\!::e/\uparrow\!::c/\uparrow\!::c/\uparrow\!::s \tag{10}$$

In (7), labels $c_{i_1}, \cdots, c_{i_n}$ represent the clauses in ϕ that contain positive literal x_i (the clauses becoming true by setting $x_i = true$). We assume that, if ϕ contains only $k < n$ such clauses, then $c_{i_{k+1}} = \cdots = c_{i_n} = c_{n+1}$, where c_{n+1} is a "dummy". For example, if $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, $i = 1$, and x_1 occurs in C_1 and C_3, then we obtain the following

$$/y/c_1/\uparrow\!::y/y/c_3/\uparrow\!::y/y/c_5/\uparrow\!::y/y/c_5/\uparrow\!::y/b',$$

where labels c_1 and c_3 represent clauses C_1 and C_3, respectively, and c_5 is a dummy label. q_{T_i} works as follows.

1. By (5), nondeterministically selects one of the two c-nodes in (A) of Fig. 6.
2. By (6), moves down to the x-node at either (x_i) or (x'_i) in Fig. 6.
3. By (7), "constructs" a subtree t_{T_i} (Fig. 7 (a)) as the left subtree of the x-node in step 2 above, then by (8) goes back to the x-node.
4. By (9) and (10), goes back to the root node labeled by s.

q_{F_i} is defined similarly to q_{T_i}, as follows.

$$q_{F_i} = /\downarrow^*\!::c \tag{11}$$

$$\underbrace{/x/ \;\cdots\; /x}_{i\,\text{location steps}}$$

$$/y/c'_{i_1}/\uparrow\!::y/y/c'_{i_2}/\uparrow\!::y/\cdots/y/c'_{i_n}/\uparrow\!::y/b' \tag{12}$$

$$\underbrace{/\uparrow\!::y/\cdots/\uparrow\!::y}_{n\text{ location steps}}$$

$$\underbrace{/\uparrow\!::x/\cdots/\uparrow\!::x}_{i\text{ location steps}}$$

$$/\uparrow\!::c/\downarrow^*\!::e/\uparrow\!::c/\uparrow\!::c/\uparrow\!::s$$

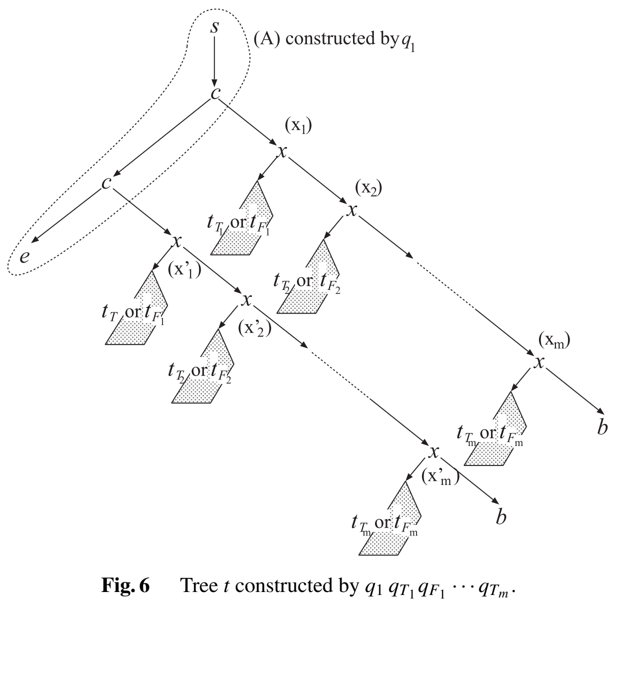


Fig. 6 Tree t constructed by $q_1\, q_{T_1} q_{F_1} \cdots q_{T_m}$.

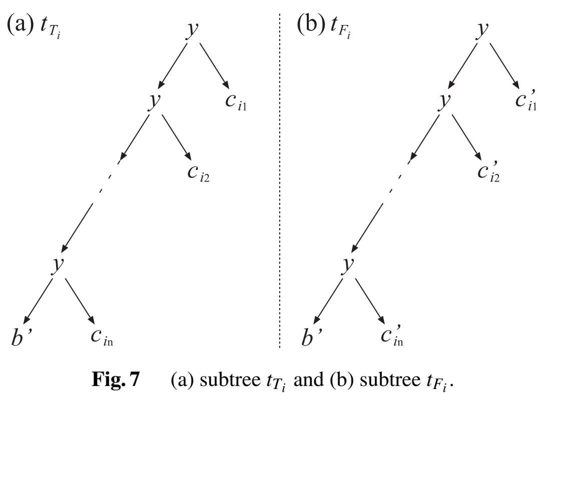


Fig. 7 (a) subtree t_{T_i} and (b) subtree t_{F_i}.

In (12), $c'_{i_1}, \cdots, c'_{i_n}$ represent the clauses in ϕ that contain negative literal $\neg x_i$. If ϕ contains only $k < n$ such clauses, then we set $c'_{i_{k+1}} = \cdots = c'_{i_n} = c_{n+1}$. q_{T_i} works similarly to q_{T_i}, except that q_{F_i} constructs a subtree t_{F_i} instead of t_{T_i} as the left subtree of the x-node at (x_i) or (x'_i) (Fig. 7 (b)). Note that by the assumption $\{c_{i_1}, \cdots, c_{i_n}\} \cap \{c'_{i_1}, \cdots, c'_{i_n}\} \subseteq \{c_{n+1}\}$. This implies that the labels of leaf nodes in t_{T_i} must be different from those of t_{F_i}, i.e., t_{T_i} and t_{F_i} cannot "overlap". Therefore, the c-nodes selected by q_{T_i} and q_{F_i} ((5) and (11)) must be distinct for every $1 \le i \le m$.

Let us define q_2 and q_{check_i}. First, q_2 selects the upper c-node in (A) of Fig. 6 and moves to its right child (the x-node at (x_1)).

$$q_2 = /c/x$$

Then q_{check_i} is defined as follows ($1 \le i \le n$).

$$\begin{aligned} q_{check_i} = {} & /{\downarrow^*}{::}\, c_i && (13)\\ & /{\uparrow}{::}\, y/{\downarrow^*}{::}\, b' && (14)\\ & \underbrace{/{\uparrow}{::}\, y/\cdots/{\uparrow}{::}\, y}_{n \text{ location steps}}/{\uparrow}{::}\, x && (15)\\ & /{\downarrow^*}{::}\, b\underbrace{/{\uparrow}{::}\, x/\cdots/{\uparrow}{::}\, x}_{m \text{ location steps}} && (16) \end{aligned}$$

By (13), q_{check_i} checks if the x-node at (x_1) has a descendant labeled by c_i, then by (14)-(16) goes back to the x-node. Therefore, $q_2\, q_{check_1} \cdots q_{check_n}$ checks whether the x-node at (x_1) has a descendant labeled by c_i for every $1 \le i \le n$.

Now we show that ϕ is satisfiable *iff* q is satisfiable under D.

Only if part: Assume that ϕ is satisfiable. Then there is a truth assignment α for $x_1, \cdots, x_m$ that satisfies ϕ. We can construct a tree t as shown in Fig. 6 that satisfies the following conditions for every $1 \le i \le m$.

- If $\alpha(x_i) = true$, the left subtree of the the x-node at (x_i) is t_{T_i}.
- If $\alpha(x_i) = false$, the left subtree of the the x-node at (x_i) is t_{F_i}.

Since α satisfies ϕ, the x-node at (x_1) has a descendant c_i for every $1 \le i \le n$. Hence t passes the check of $q_2 q_{check_1}, \cdots, q_{check_n}$.

If part: Assume that q is satisfiable under D. Then there is a tree t valid against D such that $t \vDash q$. By the construction of q and D, t is of the form presented in Fig. 6 and satisfies $q_2, q_{check_1}, \cdots, q_{check_n}$. Thus, the x-node at (x_1) has a descendant c_i for every $1 \le i \le n$. It is easy to show that the following truth assignment α satisfies ϕ.

$$\alpha(x_i) = \begin{cases} true & \text{if the left subtree of the } x\text{-node} \\ & \text{at } (x_i) \text{ is } t_{T_i}, \\ false & \text{if the left subtree of the } x\text{-node} \\ & \text{at } (x_i) \text{ is } t_{F_i}. \end{cases}$$

□

6. Algorithm for XPath Fragment with Descendant-or-Self Axis

In this section, we present an algorithm for solving SAT(XP$^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+,\downarrow^*\}}$) under duplicate-free DTDs.

We first extend the definition of traverse tree so that it can handle descendant-or-self axes. We use dashed edge $n' \dashrightarrow n$, which means that n is a descendant of n'. Formally, a *traverse tree* of $q \in$ XP$^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ is defined as follows.

- The case where $|q| = 1$: Let $q = /ax :: lb$.
  - If $ax = \downarrow$, then an edge $n' \rightarrow n$ such that $l(n') = root$, $l(n) = lb$, $I(n') = \emptyset$, and that $I(n) = \{1\}$ is the traverse tree of q.
  - If $ax = \downarrow^*$, then an edge $n' \dashrightarrow n$ such that $l(n') = root$, $l(n) = lb$, $I(n') = \emptyset$, and that $I(n) = \{1\}$ is the traverse tree of q.
- The case where $|q| > 1$: Let $q = q'/ax :: lb$, where q' is a query with $|q'| = |q| - 1$. Let t be a traverse tree of q' and n be the "context node" in t, i.e., $|q'| \in I(n)$. We have five cases according to ax.
  1. The case where $ax = \downarrow$: Any tree obtained from t by Case (1) of the previous definition in Sect. 4 is a traverse tree of q.
  2. The case where $ax = \downarrow^*$:
     a. If $l(n') = lb$ for some descendant n' of n, then the tree obtained from t by adding $|q|$ to $I(n')$ is a traverse tree of q, or
     b. Let n' be a descendant in t. The tree obtained by adding an edge $n' \dashrightarrow n_{new}$ to t is a traverse tree of q, where n_{new} is a new node with $l(n_{new}) = lb$ and $I(n_{new}) = \{|q|\}$.
  3. The case where $ax = \uparrow$:
     a. If the edge entering n is not a dashed edge, then the tree obtained from t by Case (2) of the previous definition in Sect. 4 is a traverse tree of q.
     b. Otherwise, let $n' \dashrightarrow n$ be the edge entering n.
        i. If $l(n') = lb$, then the tree obtained from t by replacing $n' \dashrightarrow n$ with $n' \rightarrow n$ and adding $|q|$ to $I(n')$ is a traverse tree of q, or
        ii. The tree obtained from t by replacing $n' \dashrightarrow n$ with a path $n' \dashrightarrow n_{new} \rightarrow n$ is a traverse tree of q, where n_{new} is a new node with $l(n_{new}) = lb$ and $I(n_{new}) = \{|q|\}$.
  4. The case where $ax = \rightarrow^+$:
     a. If the edge entering n is not a dashed edge, then any tree obtained by Case (3) of the previous definition in Sect. 4 is a traverse tree of q.

b. Otherwise, let t' be a tree obtained by modifying t by Case (3-b) above. The tree obtained from t' by adding a new node n_{new} as a right sibling of n is a traverse tree of q, where $l(n_{new}) = lb$ and $I(n_{new}) = \{|q|\}$.

5. The case of $ax = \leftarrow^+$ is defined similarly to Case (4) above.

The validity of a traverse tree t of $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ against D is defined similarly to Sect. 4, except that any dashed edge in t is replaced by an appropriate path containing no dashed edge.

We extend the algorithm in Sect. 4 so that it handles descendant-or-self axes. Let t be a traverse tree, n be the marked node in t, and l be a label. There may be more than one node reachable from n via location step $\downarrow^*:: l$, and we have to check the satisfiability for each such node. Accordingly, for each such node n' the algorithm makes a copy t' of t and mark the node corresponding to n' in t', then check the satisfiability of each copied tree. We use a variable T to hold a set of such trees, and the algorithm returns "unsatisfiable" if T becomes empty. Also, each subroutine returns a set of trees obtained by modifying input tree t according to location step axis[i] :: label[i]. Let us first present the main algorithm.

```
Input: A query $q$ = /axis[1] ::label[1]/ ⋯ /axis[$m$] ::label[$m$]
       and a duplicate-free DTD $D = (d, s)$.
Output: "satisfiable" or "unsatisfiable".
  begin
 1.  Create a node $n$ labeled by "root".
     Let $t$ be the tree consisting only of $n$.
     Assume that $d$(root) = $s$.
 2.  Mark $n$.
 3.  $T \leftarrow \{t\}$;
 4.  for $i = 1$ to $m$ do
 5.    $T' \leftarrow \emptyset$;
 6.    for each $t \in T$ do
 7.      if axis[$i$] = '$\downarrow$' then
 8.        $T' \leftarrow T' \cup$ do_child'($t$, label[$i$]);
 9.      else if axis[$i$] = '$\downarrow^*$' then
10.        $T' \leftarrow T' \cup$ do_descendant-or-self'($t$, label[$i$]);
11.      else if axis[$i$] = '$\uparrow$' then
12.        $T' \leftarrow T' \cup$ do_parent'($t$, label[$i$]);
13.      else if axis[$i$] = '$\rightarrow^+$' then
14.        $T' \leftarrow T' \cup$ do_following-sibling'($t$, label[$i$]);
15.      else if axis[$i$] = '$\leftarrow^+$' then
16.        $T' \leftarrow T' \cup$ do_preceding-sibling'($t$, label[$i$]);
17.      end
18.    end
19.    $T \leftarrow T'$;
20.    if $T = \emptyset$ then
21.      return "unsatisfiable";
22.    end
23.  end
24.  return "satisfiable";
  end
```

Let us present the subroutines in lines 8 to 16. First, do_child$'$ just calls do_child defined in Sect. 4.

```
do_child'($t, l$)
  begin
 1.  $t \leftarrow$ do_child($t, l$);
 2.  if $t \neq nil$ then
 3.    return $\{t\}$;
 4.  end
 5.  return $\emptyset$;
  end
```

To give do_descendant-or-self$'$, we need some definitions. Let t be a tree, n be a node in t, and n' be a descendant of n. By $L_t(n, n')$, we mean the sequence of labels on the path from n to n' in t. For example, $L_t(n_2, n_4) = bcd$ in Fig. 9 (a). Let p be a path. By $tail(p)$ we mean the last node of p. By $t +_{n'} p$ we mean the tree obtained by appending p to t at n' (Fig. 9 (b)). For a sequence L of labels, n' is a *potential branch point* w.r.t. (t, n, L) if there is a path $p = n_1 \to \cdots \to n_k$ such that n_1 is insertable as a child of n', $d(l(n_i))$ contains $l(n_{i+1})$ for every $1 \le i \le k-1$, and that $L_{t'}(n, tail(p)) = L$, where $t' = t +_{n'} p$. For example, in Fig. 9 (b) n_3 is a potential branch point w.r.t. $(t, n_2, bcabd)$.

do_descendant-or-self$'(t, l)$ returns a set of trees obtained by the following (a) and (b).

(a) Let n be the marked node in t. We have to consider the descendants of n labeled by l in t. Thus, for each descendant n' of n labeled by l, we make a copy t' of t and mark the node corresponding to n' in t'.
(b) We also have to consider nodes "outside" t. Let L be a sequence of labels whose last label is l, n' be a potential branch point w.r.t. (t, n, L), and p be a path such that $L = L_{t +_{n'} p}(n, tail(p))$. Then $tail(p)$ is potentially a descendant of n labeled by l. To remember such a node, we make a new node n_{new} labeled by l and add a dashed edge $n' \dashrightarrow n_{new}$ to t.

Now let us present do_descendant-or-self$'$. (a) is done in lines 3 to 8 and (b) is done in lines 9 to 17.

```
do_descendant-or-self'($t, l$)
  begin
 1.  $T' \leftarrow \emptyset$;
 2.  Let $n$ be the marked node in $t$.
 3.  $N \leftarrow \{n' \mid n'$ is a descendant of $n$ labeled by $l\}$;
 4.  for each $n' \in N$ do
 5.    Create a copy $t'$ of $t$.
 6.    Mark the node corresponding to $n'$ in $t'$.
 7.    Add $t'$ to $T'$.
 8.  end
 9.  $B \leftarrow \{n' \mid L$ is a sequence of labels whose last label is
         $l$, $n'$ is a potential branch point w.r.t. $(t, n, L)\}$;
10.  if $B \neq \emptyset$ then
11.    Create a new node $n_{new}$ labeled by $l$.
12.    for each $n' \in B$ do
13.      Add a dashed edge $n' \dashrightarrow n_{new}$ to $t$.
14.    end
15.    Mark $n_{new}$.
16.    Add $t$ to $T'$.
17.  end
18.  return $T'$;
  end
```

For example, let

$$q = \underbrace{/a/b/\uparrow:: a/c/f}_{q'} /\uparrow:: c/ \uparrow:: a/\downarrow^*:: f/\uparrow:: b/\rightarrow^+:: g$$

and $D = (d, a)$ be a DTD, where $d(a) = f?bc$, $d(b) = f|(bg?)$, $d(c) = f$, $d(f) = d(g) = \epsilon$. Let t be the tree shown in

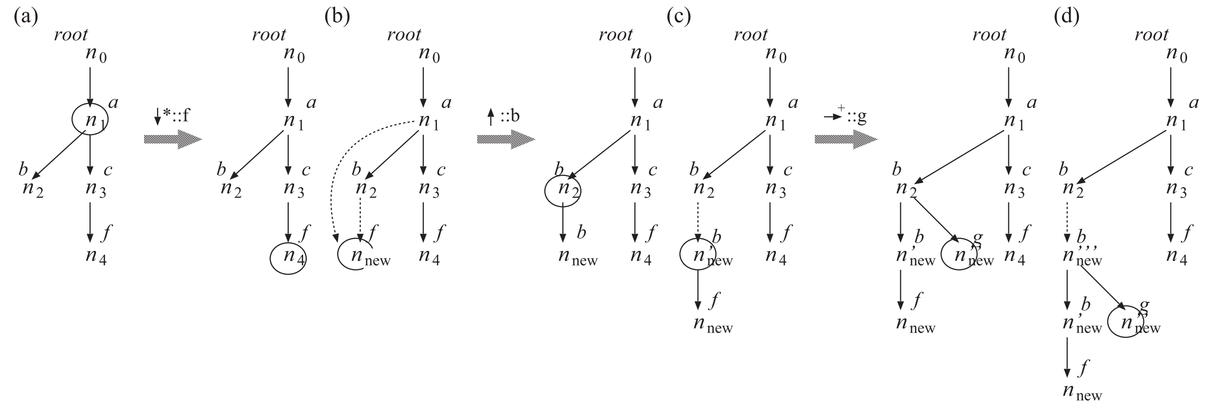


Fig. 8 Trees created by the algorithm in Sect. 6.

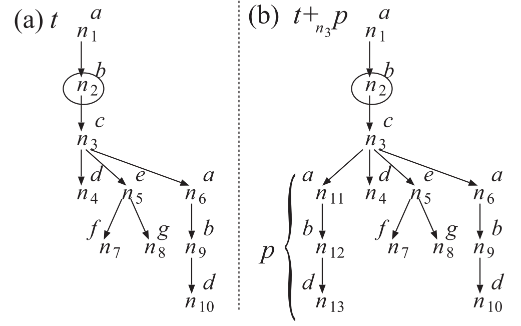


Fig. 9 An example of a potential branch point.

Fig. 8 (a), which is constructed by the algorithm for (q', D). Then do_descendant-or-self$'(t, f)$ returns a set of two trees shown in Fig. 8 (b). In lines 3 to 8 the left tree in Fig. 8 (b) is created. Consider lines 9 to 17. Among the descendants of n_1 in t, n_1, n_2, and n_3 can have a descendant labeled by f due to D. n_1 and n_2 are potential branch points w.r.t. (t, n_1, af) and (t, n_1, abf), respectively, but n_3 is not since n_3 already has a child n_4 labeled by f. Thus we obtain $B = \{n_1, n_2\}$ and the right tree in Fig. 8 (b) is created.

We next present do_parent$'$. Let n be the marked node in t. If there is no dashed edge entering n, then it suffices to call do_parent defined in Sect. 4. Otherwise, let $n' \dashrightarrow n$ be a dashed edge entering n. do_parent$'$ does the following.

(a) If n' is labeled by l and n is insertable as a child of n', then n' can be the parent n. Thus $n' \dashrightarrow n$ is replaced by $n' \rightarrow n$ and n' is marked.
(b) If a node labeled by l can be a proper descendant of n' as well as the parent of n, then a new node n_{new} labeled by l is inserted as the parent of n, that is, $n' \dashrightarrow n$ is "expanded" to $n' \dashrightarrow n_{new} \rightarrow n$.

Let us present do_parent$'$. (a) is done in lines 9 to 14 and (b) is done in lines 15 to 21. In line 15, we say that a dashed edge $n' \dashrightarrow n$ can be *expanded* by a node labeled by l if we can construct a path $n_1 \rightarrow \cdots \rightarrow n_{k-1} \rightarrow n_k$ such that $l(n_1) = l(n')$, $l(n_{k-1}) = l$, $l(n_k) = l(n)$, and that $d(l(n_i))$ contains $l(n_{i+1})$ for $1 \le i \le k-1$.

do_parent$'(t, l)$

```
begin
1.  $T' \leftarrow \emptyset$;
2.  Let $n$ be the marked node in $t$.
3.  **if** there is no dashed edge entering $n$ **then**
4.      $t \leftarrow$ do_parent$(t, l)$;
5.      **if** $t \neq nil$ **then**
6.          Add $t$ to $T'$.
7.      **end**
8.  **else**
9.      **for each** dashed edge $n' \dashrightarrow n$ such that $n'$ is labeled
        by $l$ and that $n$ is insertable as a child of $n'$ **do**
10.         Create a copy $t'$ of $t$.
11.         Replace $n' \dashrightarrow n$ by $n' \rightarrow n$ in $t'$.
12.         Delete every dashed edge entering $n$ in $t'$.
13.         Add $t'$ to $T'$.
14.     **end**
15.     **if** $t$ contains a dashed edge $n' \dashrightarrow n$ that can be
        expanded by a node labeled by $l$ **then**
16.         Delete every dashed edge $n' \dashrightarrow n$ of $t$ that cannot
            be expanded by a node labeled by $l$.
17.         Create a new node $n_{new}$ labeled by $l$.
18.         Insert $n_{new}$ as the parent of $n$.
19.         Mark $n_{new}$.
20.         Add $t$ to $T'$.
21.     **end**
22. **end**
23. **return** $T'$;
end
```

For example, let t_l and t_r be the left and right trees in Fig. 8 (b), respectively. The result of do_parent$'(t_l, b)$ is empty, since the parent of n_4 in t_l is labeled by c and nil is returned in line 4. On the other hand, do_parent$'(t_r, b)$ returns a set of the two trees shown in Fig. 8 (c); (i) by replacing $n_2 \dashrightarrow n_{new}$ with $n_2 \rightarrow n_{new}$ in line 11 the left tree in Fig. 8 (c) is obtained, and (ii) by inserting a new node n'_{new} as the parent of n_{new} in line 18 the right tree in Fig. 8 (c) is obtained. In both cases, dashed edge $n_1 \dashrightarrow n_{new}$ is deleted.

Finally, let us present do_following-sibling$'$. Let n be the marked node in t. If there is no dashed edge entering n, then it suffices to call do_following-sibling defined in Sect. 4. Assume that there is a dashed edge entering n. We have to first identify the parent of n, and then create a right sibling of n labeled by l. Thus, (a) do_following-sibling$'$ first constructs trees in which the parents of n are identified (lines 9 to 12), then (b) for each tree t'' found in (a) the subroutine inserts a new node labeled by l as a right sibling of n (lines 13 to 18). In line 9 the subroutine finds the labels that can be the parent of n.

```
do_following-sibling'(t, l)
  begin
1.  $T' \leftarrow \emptyset$;
2.  Let $n$ be the marked node in $t$.
3.  if there is no dashed edge entering $n$ then
4.    $t \leftarrow$ do_following-sibling($t, l$);
5.    if $t \neq nil$ then
6.      Add $t$ to $T'$.
7.    end
8.  else
9.    $L \leftarrow \{l' \mid w_1 l(n) w_2\, l\, w_3 \in L(d(l'))$ for some words
          $w_1, w_2, w_3\}$.
10.   for each $l' \in L$ do
11.     Create a copy $t'$ of $t$.
12.     $T'' \leftarrow$ do_parent'($t', l'$);
13.     for each $t'' \in T''$ do
14.       Create a new node $n_{new}$ labeled by $l$.
15.       Let $n'$ be the node in $t''$ corresponding to $n$.
16.       Add $n_{new}$ to $t''$ as a right sibling of $n'$.
17.       Mark $n_{new}$.
18.       Add $t''$ to $T'$.
19.     end
20.   end
21. end
22. return $T'$;
  end
```

For example, let t'_l and t'_r be the left and right trees in Fig. 8 (c), respectively. do_following-sibling$'(t'_l, g)$ returns an empty set, since n_2 cannot have any sibling labeled by g due to $d(a) = f?bc$ and thus do_following-sibling(t'_l, g) returns *nil* in line 4. Consider do_following-sibling$'(t'_r, g)$. In line 9, we obtain $L = \{b\}$. In line 12, we obtain two trees shown in Fig. 10, and for each of the trees a new node labeled by g is inserted as the right sibling of n'_{new} (Fig. 8 (d)).

Example 1: Let $q = /{\downarrow^*}::b/{\rightarrow^+}::c/{\uparrow}::a$ and $D = (d, s)$ be a DTD, where $d(s) = a|a'$, $d(a) = d(a') = bc$, $d(b) = d(c) = \epsilon$. Figure 11 presents the trees created by the algorithm for each location step of q. Since $L = \{a, a'\}$ in line 9 of do_following-sibling$'$, two trees are created by location step "$\rightarrow^+$:: c". Then the right tree of the two trees is deleted according to location step "$\uparrow$:: a" since $l(n_{new}) \neq a$. □

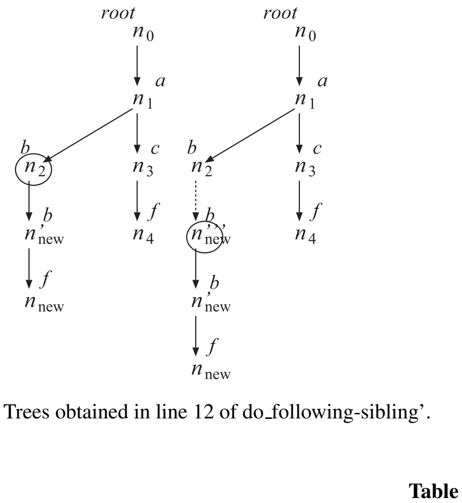


Fig. 10 Trees obtained in line 12 of do_following-sibling'.

We have the following results (the proofs are shown in the Appendix).

Theorem 6: Let $q \in XP^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ be a query and D be a duplicate-free DTD. Then the algorithm returns "satisfiable" *iff* q is satisfiable under D. □

Theorem 7: Let $q \in XP^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ be a query, D be a duplicate-free DTD, and c be a constant number. If q contains at most c descendant-or-self axes, then the algorithm runs in $O((|q| \cdot |\Sigma|)^c \cdot |q|^2 \cdot |D|^2)$ time. □

Thus, under duplicate-free DTDs, the algorithm runs in polynomial time if the number of descendant-or-self axes in a query is bounded by a constant. Here, suppose that we have no restriction on D. Then by Theorems 1 and 2 it is unlikely that the algorithm runs in polynomial time even if the number of descendant-or-self axes in a query is constant.

7. Conclusion

In this paper, we first showed that SAT($XP^{\{\downarrow,\uparrow\}}$) and SAT($XP^{\{\downarrow,\rightarrow^+,\leftarrow^+\}}$) are NP-complete under non-restricted DTDs but that SAT($XP^{\{\downarrow,\uparrow,\rightarrow^+,\leftarrow^+\}}$) is in PTIME under duplicate-free DTDs. We next showed that SAT($XP^{\{\downarrow,\downarrow^*,\uparrow\}}$) is NP-complete under duplicate-free DTDs and proposed an algorithm for solving SAT($XP^{\{\downarrow,\downarrow^*,\uparrow,\rightarrow^+,\leftarrow^+\}}$) under duplicate-free DTDs.

However, there are many things to do as future works. First, this paper presents no experimental result. Thus we need to implement the algorithms and conduct experiments to examine the efficiency of our algorithm. Second, this paper considered only DTDs as a schema language. It is important to consider the satisfiability problem under more powerful schema languages such as regular tree grammar.

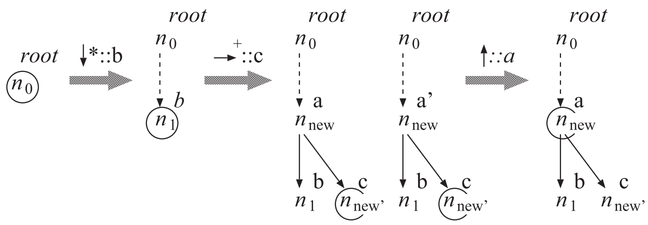


Fig. 11 A tree "cancelled" by an upward axis.

Table 1 Syntax of XP′.

XP′	::=	"/" LocationStep | "/" LocationStep XP′
LocationStep	::=	Axis "::" Label | Axis "::" Label PredSequence
Axis	::=	"↓" | "↑" | "↓*" | "→+" | "←+"
Label	::=	(any label in Σ)
PredSequence	::=	"[" RelativePath "]" | "[" RelativePath "]" PredSequence
RelativePath	::=	ChildLocationStep | ChildLocationStep "/" RelativePath
ChildLocationStep	::=	"↓"::Label | "↓"::Label PredSequence

Third, The XPath fragments considered so far are restricted in the sense that only a label is allowed as a node test and no qualifier is supported. This restriction can be relaxed slightly. If a query q has qualifiers using only child axes, then q can be rewritten into an equivalent query without qualifier. For example, consider the following query

$$/a/b[c/d[e]]/f$$

This can be rewritten into the following equivalent one.

$$/a/b/c/d/e/\uparrow :: d/\uparrow :: c/\uparrow :: b/f.$$

Thus, our algorithms can be applied to a query in a more general XPath fragment, formally shown as XP′ in Table 1. XP′ is the same as XP in Sect. 2 except that XP′ can have qualifiers using only child axes.

References

[1] M. Benedikt, W. Fan, and F. Geerts, "XPath satisfiability in the presence of dtds," J. ACM, vol.55, no.2, 2008.
[2] F. Geerts and W. Fan, "Satisfiability of XPath queries with sibling axes," Proc. DBPL, pp.122–137, 2005.
[3] M. Montazerian, P.T. Wood, and S.R. Mousavi, "XPath query satisfiability is in PTIME for real-world DTDS," Proc. XSym, pp.17–30, 2007.
[4] J. Hidders, "Satisfiability of XPath expressions," Proc. DBPL, pp.21–36, 2003.
[5] L.V.S. Lakshmanan, G. Ramesh, H. Wang, and Z.J. Zhao, "On testing satisfiability of tree pattern queries," Proc. VLDB, pp.120–131, 2004.
[6] D. Figueira, "Satisfiability of downward XPath with data equality tests," Proc. PODS, pp.197–206, 2009.
[7] Y. Ishihara, T. Morimoto, S. Shimizu, K. Hashimoto, and T. Fujiwara, "A tractable subclass of DTDs for XPath satisfiability with sibling axes," Proc. DBPL, pp.68–83, 2009.
[8] Y. Ishihara, S. Shimizu, and T. Fujiwara, "Extending the tractability results on XPath satisfiability with sibling axes," Proc. XSym, pp.33–47, 2010.
[9] N. Suzuki and Y. Fukushima, "Satisfiability of simple XPath fragments in the presence of DTDs," Proc. WIDM, pp.15–22, 2009.
[10] G. Gottlob, C. Koch, and R. Pichler, "The complexity of XPath query evaluation," Proc. PODS, pp.179–190, 2003.

Appendix: Proofs for Theorems 6 and 7

Theorem 6 can be proved by three steps similar to the proof for Theorem 3. First, the following lemma holds similarly to Lemma 1.

Lemma 3: Let $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ be a query and D be a duplicate-free DTD. Then there is a traverse tree of q valid against D *iff* q is satisfiable under D. □

Then the following lemma corresponds to Lemma 2. Since the algorithm in this section constructs a set of trees instead of a single tree, the following lemma is extended accordingly.

Lemma 4: Let D be a duplicate-free DTD, $q \in \mathrm{XP}^{\{\downarrow,\uparrow,\downarrow^*,\rightarrow^+,\leftarrow^+\}}$ be a query, and T be the set of trees obtained by the algorithm for (q, D). Then the following holds.

1. For any $t \in T$, t is a traverse tree of q valid against D.
2. For every traverse tree t' of q valid against D, there is a tree $t \in T$ such that there is a total and surjective function $h : N \rightarrow N'$ satisfying the following, where N and N' are the sets of nodes of t and t', respectively.
   - For every node n' in t', $I(n') = I(n_i) \cup \cdots \cup I(n_j)$, where $n_i, \cdots, n_j$ is the nodes in t such that $h(n_i) = \cdots = h(n_j) = n'$.

Proof: Condition (1) follows from the construction of the algorithm. We show by induction on $|q|$ that Condition (2) holds.

Basis: $|q| = 1$. Let $q = /ax :: lb$, where $ax \in \{\downarrow, \downarrow^*\}$ and lb is a label. Suppose first that $ax =\downarrow$. Let $t = n' \rightarrow n$, where $l(n') = root$, $l(n) = lb$, $I(n') = \emptyset$, and $I(n) = \{1\}$. If $lb = s$, then t is the traverse tree of q valid against D and $T = \{t\}$, thus Condition (2) holds. Otherwise, there is no traverse tree of q valid against D by definition. Suppose next that $ax =\downarrow^*$ and let $t = n' \dashrightarrow n$, where $l(n') = root$, $l(n) = lb$, $I(n') = \emptyset$, and $I(n) = \{1\}$. If lb is reachable from s in D, then t is the traverse tree of q valid against D, and $T = \{t\}$ by the construction of the algorithm. Here, for labels $a, b \in \Sigma$, b is *reachable* from a if (i) $a = b$ or (ii) b occurs in $d(c)$ for some label c reachable from a. Otherwise, there is no traverse tree of q valid against D.

Induction: Assume as an induction hypothesis that if $|q| < i$, then Condition (2) holds. Consider the case where $|q| = i$. Let $q = q'/ax :: lb$, where q' is a query with $|q'| = i-1$. Let $T_{q'}$ be the set of trees obtained by the algorithm for (q', D). Moreover, let t' be a traverse tree of q' valid against D, and let n be the "context node" in t', i.e., $|q'| \in I(n)$. We have five cases according to ax. In the following, we show the cases where $ax =\downarrow$ and $ax =\uparrow$ (the other cases can be shown similarly).

The case where $ax =\downarrow$: If a node labeled by lb is not insertable as a child of n and n has no child labeled by lb, then any traverse tree of q obtained from t' is invalid against D. Conversely, suppose first that a node labeled by lb is insertable as a child of n. Let t'_2 be the tree obtained from t' by inserting a node labeled by lb as a child of n. Moreover, let t be the tree in $T_{q'}$ "corresponding" to t', i.e., t is the tree such that Condition (2) holds for t and t' (t must be exist by the induction hypothesis), and t_2 be the tree obtained from t by inserting a node labeled by lb as a child of the node n_2 corresponding to n, i.e., $h(n_2) = n$. Then by the construction of the algorithm we can show that $t_2 \in T$ and that Condition (2) holds for t_2 and t'_2 since Condition (2) holds for t and t'. Suppose next that a node labeled by lb is not insertable as a child of n but n has a child labeled by lb. In this case, we can show that Condition (2) holds similarly to above.

The case where $ax =\uparrow$: Suppose first that the edge entering n is not a dashed edge. Let n' be the parent of n in t', and let t'_2 be the tree obtained from t' by adding i to $I(n')$. If $lb \neq l(n')$, then by definition t'_2 is not a traverse tree of q.

Otherwise, t_2' is a traverse tree of q valid against D. Let t be the tree in $T_{q'}$ corresponding to t' (t must be exist by the induction hypothesis), and let t_2 be the tree obtained from t by adding i to $I(n_2)$, where n_2 is the node corresponding to n', i.e., $h(n_2) = n'$. Then by the construction of the algorithm we can show that $t_2 \in T$ and that Condition (2) holds for t_2 and t_2' since Condition (2) holds for t and t'. Suppose next that the edge entering n is a dashed edge. If (i) n is not insertable as a child of n' and (ii) $n' \dashrightarrow n$ cannot be expanded by a node labeled by lb, then any tree obtained from t' by Case (3) cannot be valid against D. Consider first (i) and suppose that n is insertable as a child of n', and let t_2' be the tree obtained from t' by replacing $n' \dashrightarrow n$ with $n' \rightarrow n$. Moreover, let t be the tree in $T_{q'}$ corresponding to t' and t_2 be the tree obtained from t by replacing $n_2' \dashrightarrow n_2$ with $n_2' \rightarrow n_2$, where $h(n_2') = n'$ and $h(n_2) = n$. Then by the construction of the algorithm we can show that $t_2 \in T$ and that Condition (2) holds for t_2 and t_2' since Condition (2) holds for t and t'. Second, (ii) can be shown in a similar way. □

Now Theorem 6 can be shown as follows. First, if the algorithm returns "satisfiable", then it is easy to show that the tree created by the algorithm is a traverse tree valid against D, with each "shared" node created in lines 11 to 14 of do_descendant-or-self′ extracted. Suppose next that the algorithm returns "unsatisfiable". Then $T = \emptyset$ for (q, D). This implies that there is no traverse tree of q valid against D by Condition (2) of Lemma 4. Thus q is unsatisfiable under D by Lemma 3.

Consider next Theorem 7. Assume that q contains c descendant-or-self axes. Let us first consider the number of trees created by the algorithm. Since q contains c descendant-or-self axes, we can denote

$$q = /p_0/{\downarrow^*}{::}\, l_1/p_1/{\downarrow^*}{::}\, l_2/p_2/\cdots/{\downarrow^*}{::}\, l_c/p_c,$$

where p_i is a sequence of zero or more location steps using $\downarrow$, $\uparrow$, $\rightarrow^+$, and $\leftarrow^+$ axes and l_i is a label. For $1 \le i \le c$, let k_i be the number of $\rightarrow^+$ and $\leftarrow^+$ axes in p_i. Then p_i can be denoted

$$p_i = /p_{i,1}/s_{i,1}/p_{i,2}/\cdots/s_{i,k_i}/p_{i,k_i+1},$$

where $p_{i,j}$ is a sequence of zero or more location steps using $\downarrow$ and $\uparrow$ axes, and $s_{i,j}$ is a location step whose axis is $\rightarrow^+$ or $\leftarrow^+$.

We show that due to p_i the number of trees kept by the algorithm is increased by a factor of $O(|\Sigma|)$. In the following, we consider the case where $i = 1$ (the other cases can be shown similarly). For an index j with $1 \le j \le k_1$, suppose that the algorithm has a tree $t_{1,j}$ when the algorithm tries to process $s_{1,j} \in \{\rightarrow^+, \leftarrow^+\}$. Let n_k be the marked node in $t_{1,j}$. If no dashed edge enters n_k, then no new tree is created by do_following-sibling′/do_preceding-sibling′. On the other hand, if the edge entering n_k is a dashed edge, then $O(|\Sigma|)$ trees are created by do_following-sibling′/do_preceding-sibling′ (see lines 9 to 12 of do_following-sibling′). Let p_1' be the subquery of p_1 following $s_{1,j}$, that is,

$$p_1' = /p_{1,j+1}/s_{1,j+1}/p_{1,j+2}/\cdots/s_{1,k_1}/p_{1,k_1+1}.$$

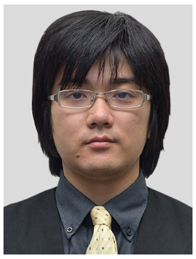
If none of the ancestors of n_k is marked during processing p_1', then no new tree is created since at any time the edge entering the marked node cannot be a dashed edge. Suppose on the other hand that the parent of n_k is marked by processing some upward location step in p_1', say $\uparrow{::}\, l$. Then all the trees created from $t_{1,j}$ as above except one are deleted by do_parent′ since the subroutine deletes every tree such that the parent of n_k is not labeled by l (lines 3 to 7 of do_parent′). That is, as shown above by $s_{1,j}$ $O(|\Sigma|)$ trees are created but the increase by $s_{1,j}$ is "cancelled" by the upward location step (see Example 1 below). A similar argument can be applied to the rest of location steps in p_1'. Therefore, from one tree $O(|\Sigma|)$ new trees are created by processing p_1.

For a tree t and a location step ${\downarrow^*}{::}\, l_i$, $O(|t|)$ trees are created by do_descendant-or-self′(t, l_i), and the size of any tree created by the algorithm is bounded by the number of location steps in q. Thus $O((|q| \cdot |\Sigma|)^c)$ trees are created by processing q.

Finally, do_descendant-or-self′ is the most time-consuming subroutine and let us consider its time complexity. In the subroutine, computing the set B in line 9 is the most complex. To obtain B, we first find set $R(l) = \{l' \mid l \text{ is reachable from } l' \text{ in } D\}$, which takes $O(|D|)$ time. Then for a descendant n' of n, $n' \in B$ *iff* for some label $l' \in R(l)$, l' appears in $d(l(n'))$ and a node labeled by l' is insertable as a child of n'. Since $|R(l)| \in O(|D|)$ and whether a node labeled by l' is insertable as a child of n' can be determined in $O(|d(l(n'))|)$ time, whether $n' \in B$ can be determined in $O(|D|^2)$ time. Since the size of input tree t is in $O(|q|)$, line 9 can be done in $O(|q| \cdot |D|^2)$ time, which is the time complexity of do_descendant-or-self′ for input (t, l). Thus lines 5 to 22 of the main algorithm runs in $O((|q| \cdot |\Sigma|)^c \cdot (|q| \cdot |D|^2))$ time for each $i = 1, \cdots, m$. Consequently, the algorithm runs in $O(|q| \cdot (|q| \cdot |\Sigma|)^c \cdot (|q| \cdot |D|^2)) = O((|q| \cdot |\Sigma|)^c \cdot |q|^2 \cdot |D|^2)$ time. Thus the theorem holds.

Nobutaka Suzuki received his bachelor's degree in information and computer sciences from Osaka University in 1993, and his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 1995 and 1998, respectively. He was with Okayama Prefectural University as a Research Associate in 1998–2004. In 2004, he joined University of Tsukuba as an Assistant Professor. Since 2009, he has been an Associate Professor of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests include database theory and structured documents.

Yuji Fukushima received his bachelor's and M.C. degrees in library and information science from University of Tsukuba in 2007 and 2009, respectively. From 2009, he has joined Yahoo Japan Corporation. His current research interests are data transformation and XPath satisfiability.



Kosetsu Ikeda received his bachelor's degree in library and information science from University of Tsukuba in 2011. He has been an M.C. student of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests are XML query processing and Web data management.